A Manual for InterComm
Version 1.0

Jae-Yong Lee, Christian Hansen, Henrique Andrade, Guy Edjlali, Alan Sussman
Department of Computer Science
University of Maryland
College Park, MD 20742
{jylee,chansen,edjlali,hcma,als}@cs.umd.edu

August 09, 2004

Contents

1 Introduction **3**

1.1 Distributions . . . 3

1.2 Linearization . . . 3

1.3 Language Interfaces . . . 4

2 Downloading and Installation **4**

3 Low-Level Programming Tasks **5**

3.1 Initializing the Library . . . 6

3.1.1 IC_Init . . . 6

3.1.2 IC_Wait . . . 6

3.1.3 IC_Sync . . . 7

3.2 Describing the Data Distribution . . . 8

3.2.1 IC_Create_bdecomp_desc . . . 8

3.2.2 IC_Create_ttable_desc . . . 9

3.2.3 IC_Translate_parti_descriptor . . . 10

3.2.4 IC_Translate_chaos_descriptor . . . 10

3.3 Defining and Communicating Array Blocks . . . 11

3.3.1 IC_Create_block_region . . . 11

3.3.2 IC_Create_enum_region . . . 11

3.3.3 IC_Compute_schedule . . . 12

3.3.4 IC_Send_TYPE . . . 13

3.3.5 IC_Recv_TYPE . . . 14

3.4 Releasing Library Resources . . . 15

3.4.1 IC_Free_program . . . 15

3.4.2 IC_Free_desc . . . 16

3.4.3 IC_Free_region . . . 16

3.4.4 IC_Free_sched . . . 17

3.4.5 IC_Quit . . . 17

4 High-Level Programming Tasks **18**

4.1 Creating Endpoints . . . 18

4.1.1 EndPoint(Constructor) . . . 18

4.2 Communicating Array Sections . . . 19

4.2.1 ExportArray . . . 19

4.2.2 ImportArray . . . 20

4.3 Termination . . . 21

4.3.1 EndPoint(Destructor) . . . 22

4.4 Error Codes . . . 22

4.4.1 PrintErrorMessage . . . 22

5 Compilation and Program Startup **23**

A Example Code **23**

A.1 Low-Level C . . . 23

A.2 Low-Level Fortran . . . 26

A.3 High-Level C++/P++ . . . 29

A.4 High-Level Fortran 90 . . . 31

1 Introduction

InterComm is a framework for coupling distributed memory parallel components that enables efficient communication in the presence of complex data distributions. In many modern scientific applications, such as physical simuations that model phenomena at multiple scales and resolutions, multiple parallel and/or sequential components need to cooperate to solve a complex problem. These components often use different languages and different libraries to parallelize their data. InterComm provides abstractions that work across these differences to provide an easy, efficient, and flexible means to move data directly from one component's data structure to another.

The two main abstractions InterComm provides are the *distribution*, which describes how data is partioned and distributed across multiple tasks (or processors), and the *linearization* which provides a mapping from one set of elements in a distribution to another.

1.1 Distributions

InterComm classifies data *distributions* into two types, those in which entire blocks of an array are assigned to tasks, a block decomposition, and those in which individual elements of an array are assigned independently to a particular task, a translation table. In the case of the former, the data structure required to describe the distribution is relatively small and can be replicated on each of the participating tasks. In the case of the latter, there is a one-to-one correspondance between the elements of the array and the number of entries in the data descriptor, therefore, the descriptor itself is rather large and must be partitioned across the participating tasks. InterComm provides two primitives for specifying these types of distrubtions (Section 3.2) as well as identifying regions for transfer within these distributions (Section 3.3).

1.2 Linearization

A *linearization* is the method by which InterComm defines an implicit mapping between the source of a data transfer distributed by one data parallel library and the destination of the transfer distributed by another library. The source and destination data elements are each described by a set of regions.

One view of the linearization is as an abstract data structure that provides a total ordering for the data elements in a set of regions. The linearization for a region is provided by the data parallel library or application writer.

We represent the operation of translating from the set of regions S_A of A, distributed by `libX`, to its linearization, L_{S_A}, by parallel library ℓ_{libX}, and the inverse operation of translating from the linearization to the set of regions as ℓ_{libX}^{-1}:

$$L_{S_A} = \ell_{libX}(S_A)$$

$$S_A = \ell_{libX}^{-1}(L_{S_A})$$

Moving data from the set of regions S_A of A distributed by `libX` to the set of regions S_B of B distributed by library `libY` can be viewed as a three-phase operation:

1. $L_{S_A} = \ell_{libX}(S_A)$
2. $L_{S_B} = L_{S_A}$
3. $S_B = \ell_{libY}^{-1}(L_{S_B})$

The only constraint on this three-phase operation is to have the same number of elements in S_A as in S_B, in order to be able to define the mapping between data elements from the source to the destination.

The concept of linearization has several important properties:

- It does not require the explicit specification of the mapping between the source data and destination data. The mapping is implicit in the separate linearizations of the source and destination data structures.
- A parallel can be drawn between the linearization and the marshal/unmarshal operations for the parameters of a remote procedure call. Linearization can be seen as an extension of the marshal/unmarshal operations to distributed data structures.

1.3 Language Interfaces

InterComm supports programs written in both C and Fortran. The functions provided for these languages are considered the *low-level* interfaces, as they provide greater flexibilty in defining distributions, but require a greater amount of attention to detail. In Section 3, the C and Fortran InterComm interfaces are presented. As a general rule, the Fortran functions are identical to their C counterparts. They only differ when a value is returned from the C function, in which case this value becomes the last parameter of the Fortran call, or when dealing with a C pointer type, which has been made to correspond with the integer type in Fortran. The C interface is specified in `intercomm.h`.

InterComm also supports programs written in Fortran 90 and C++/P++[1]. The main objective of this *high-level* interface is to encapsulate some of the complexity of describing and communicating data between the local and the remote applications. In Section 4, the C++/P++ and Fortran 90 interfaces are described.

2 Downloading and Installation

The source package, as well as an online copy of this manual and a Programmer's Reference, can be obtained from the project website at http://www.cs.umd.edu/projects/hpsl/chaos/ResearchAreas/ic/.

InterComm depends on PVM[2] for communication, so the first step is to insure that you have a working and properly configured installation of PVM available. In particular, InterComm will need the environment variables PVM_ROOT and PVM_ARCH set during configuration.

InterComm uses the GNU Autotools[3] suite for building and installation. Generally, this involves the sequence of commands:

```
./configure
make
make install
```

The `configure` command takes a number of options, use of the `--help` flag will provide a full list of those available. Options that are specific to InterComm are:

`-- with-chaos=DIR`

This causes the Chaos[4] extensions to be built (see Section 3.2.3). DIR specifies where the Chaos headers and libraries can be found.

`-- with-mbp=DIR`

This causes the MultiBlockParti[5] extensions to be built (see Section 3.2.4). DIR specifies where the MultiBlockParti headers and libraries can be found.

`-- enable-f77`

This causes the Fortran 77 interface to be built (default).

`-- enable-f90`

This causes the Fortran 90 interface to be built.

`-- with-ppp=DIR`

This causes the C++/P++ interface to be built. DIR specifies where the P++ headers and libraries can be found.

3 Low-Level Programming Tasks

This section describes how to use the C and Fortran interface for InterComm.

3.1 Initializing the Library

Before InterComm is used to transfer data between, you must first provide the runtime library with information regarding the participants. This is done with two calls, `IC_Init` and `IC_Wait`.

3.1.1 IC_Init

This function initializes the underlying communication library and creates an internal representation of the local program for use with other calls into the communication system.

Synopsis

C IC_Program* IC_Init(char* name, int tasks, int rank)

Fortran IC_Init(name, tasks, rank, myprog)
character name(*)
integer tasks, rank, myprog

Parameters

name the externally visible name of the program

tasks the number of tasks used

rank the rank of this task

Return value

an InterComm program data type

Example

```
IC_Program* myprog;
char* name = ‘‘cmpi-example’’;
int tasks = 4;
int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
myprog = IC_Init(name, tasks, rank);
```

3.1.2 IC_Wait

This function contacts a remote program to arrange for subsequent communications. It returns an internal representation the a remote program for use in data exchanging operations.

Synopsis

C IC_Program* IC_Wait(char* name, int tasks)

Fortran IC_Wait(name, tasks, prog)
character name(*)
integer tasks, prog

Parameters

name the name of the remote program

tasks the number of tasks used

Return value

an InterComm program data type

Example

```
IC_Program* prog;
char* name = ‘‘cpvm-example’’;
int tasks = 8;
prog = IC_Wait(name, tasks);
```

3.1.3 IC_Sync

This function call allows the establishment of a synchronization point between two programs by causing each to wait until both sides have made a matching call.

Synopsis

C int IC_Sync(IC_Program* myprog, IC_Program* prog)

Fortran IC_Sync(myprog, prog, status)
integer myprog, prog, status

Parameters

myprog the local program

prog the remote program

Return value

status, -1 indicates an error

Example

```
int sts;
sts = IC_Sync(myprog, prog);
```

3.2 Describing the Data Distribution

InterComm needs information about the distribution across tasks of the data structure, potentially managed by a data parallel library employed by the application. In most parallel programs manipulating large multi-dimensional arrays, a distributed data descriptor is used to store the necessary information about the regular or irregular data distribution (e.g., a distributed array descriptor for MultiBlockParti[5] or a translation table for Chaos[4]).

As the InterComm functions for moving data require a data descriptor that is meaningful within the context of InterComm, several functions are provided for describing these two types of distributions. Ideally, these functions would be used by the data parallel library developers in providing a function for translating from their descriptor type to the one used by InterComm.

3.2.1 IC_Create_bdecomp_desc

Block decompositions assign entire array sections to particular tasks, allowing for a more compact description of element locations. This function is used to describe a regular block decomposition.

Synopsis

C IC_Desc* IC_Create_bdecomp_desc(int ndims, int* blocks, int* tasks, int count)

Fortran IC_Create_bdecomp_desc(ndims, blocks, tasks, count, desc)
integer ndims, blocks(*), tasks(*), count, desc

Parameters

ndims the dimension of the distributed array

blocks a three-dimensional array of block bound specifications (see example)
The first dimension of the **blocks** array corresponds to the number of blocks used to distribute the array (one per task – see information for the **tasks** array below). The second dimension is always two (i.e., each block is represented by *two* n-dimensional points as a n-dimensional rectangular box, where n is the number of dimensions of the distributed array for which the decomposition is being created. Each of the two points represents the corners of the rectangular box). The third dimension of the **blocks** array corresponds to n, where again n is the number of the dimensions of the distributed array for which the decomposition is being created.

tasks the corresponding task assignments for the individual blocks
The k-th block in the **blocks** array (i.e., blocks[k][x][y]) is held by the k-th task (i.e., tasks[k])

count the number of blocks

Return value

an InterComm array descriptor data type

Example

```
/* 4 blocks, 2 points, 2-dimensional points */
int blocks[4][2][2] = {
  {{0,0},{3,3}}, /* block 0 */
  {{0,4},{3,7}}, /* block 1 */
  {{4,0},{7,3}}, /* block 2 */
  {{4,4},{7,7}}  /* block 3 */
};
int tasks[4] = {0,1,2,3};
IC_Desc* desc;

desc = IC_Create_bdecomp_desc(2, &blocks[0][0][0], tasks, 4);
```

3.2.2 IC_Create_ttable_desc

The translation table is a representation of an arbitrary mapping between data points and tasks in a parallel program. This information must be explicitly provided to the descriptor defining routine. This descriptor is distributed, and therefore partial for the current task. This function assigns array elements to tasks using a given partial map.

Synopsis

C IC_Desc* IC_Create_ttable_desc(int* globals, int* locals, int* tasks, int count)

Fortran IC_Create_ttable_desc(globals, locals, tasks, count, desc)
integer globals, locals(*), tasks(*), count, desc

Parameters

globals an array of global indices

locals the corresponding local indices

tasks a corresponding global index-to-task map

count the number of global indices

Return value

an InterComm array descriptor data type

Example

```
/* local portion of the global index space */
int globals[16] = {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
int locals[16] = {0,0,0,0,1,1,1,1,2,2,2,2,3,3,3,3};
int tasks[16] = {0,1,2,3,0,1,2,3,0,1,2,3,0,1,2,3};
IC_Desc* desc;

desc = IC_Create_ttable_desc(globals, locals, tasks, 16);
```

3.2.3 IC_Translate_parti_descriptor

If support for MultiBlockParti[5] is enabled during configuration (Section 2), inclusion of the header file `mbp2bdecomp.h` will allow the use a translation function for converting MultiBlockParti `DARRAY` descriptors to InterComm block decomposition descriptors.

Synopsis

C IC_Desc* IC_Translate_parti_descriptor(DARRAY* darray)

Fortran IC_Translate_parti_descriptor(darray, desc)
integer darray, desc

Parameters

darray a MultiBlockParti distributed array descriptor

Return value

an InterComm array descriptor data type

Example

```
DARRAY* darray;
IC_Desc* desc;

/* ...create DARRAY using MultiBlockParti calls... */
desc = IC_Translate_parti_desciptor(darray);
```

3.2.4 IC_Translate_chaos_descriptor

If support for Chaos[4] is enabled during configuration (Section 2), inclusion of the header file `chaos2ttable.h` will allow the use of a translation function for converting Chaos `TTABLE` descriptors to InterComm translation table descriptors.

Synopsis

C IC_Desc* IC_Translate_chaos_descriptor(TTABLE* ttable)

Fortran *There is no Fortran interface to Chaos*

Parameters

ttable a Chaos translation table descriptor

Return value

an InterComm array descriptor data type

Example

```
TTABLE* ttable;
IC_Desc* desc;

/* ...create TTABLE using Chaos calls... */
desc = IC_Translate_chaos_desciptor(ttable);
```

3.3 Defining and Communicating Array Blocks

3.3.1 IC_Create_block_region

This function allocates a region (block) designating the data elements for importing or exporting operations.

Synopsis

C IC_Region* IC_Create_block_region(int ndims, int* lower, int* upper, int* stride)

Fortran IC_Create_block_region(ndims, lower, upper, stride, region)
integer ndims, lower(*), upper(*), stride(*), region

Parameters

ndims the number of dimensions of the array

lower the lower bounds of the region to be transferred

upper the upper bounds of the region to be transferred

stride the stride of the region to be transferred

Return value

an InterComm region data type

Example

```
int ndims = 3;
int lower[3] = {0,0,0};
int upper[3] = {2,2,2};
int stride[3] = {1,1,1};
IC_Region* region_set[1];
region_set[0] = IC_Create_block_region(ndims, lower, upper, stride);
```

3.3.2 IC_Create_enum_region

This function allocates a region (enumeration) designating the data elements for importing or exporting operations.

Synopsis

C IC_Region* IC_Create_enum_region(int* indices, int size)

Fortran IC_Create_enum_region(indices, size, region)
integer indices(*), size, region

Parameters

indices a list of global indices

size the number of global indices listed

Return value

an InterComm region data type

Example

```
int indices[10] = {0,1,2,3,4,5,6,7,8,9};
int size = 10;
IC_Region* region_set[1];
region_set[0] = IC_Create_enum_region(indices, size);
```

3.3.3 IC_Compute_schedule

This function creates a communication schedule for transmitting or receiving regions (blocks) of an array. The types of array descriptors used on either end of the communication determine how and where this schedule is computed.

Synopsis

C IC_Sched* IC_Compute_schedule(IC_Program* myprog, IC_Program* prog,
IC_Desc* desc, IC_Region** region_set, int set_size)

Fortran IC_Compute_schedule(myprog, prog, desc, region_set,
set_size, sched)
integer myprog, prog, desc, region_set(*), set_size, sched

Parameters

myprog the InterComm application descriptor for the local program

prog the InterComm application descriptor for the remote program

desc the InterComm array descriptor

region_set an array of regions describing the data to be communicated

set_size the number of regions in the array

Return value

an InterComm schedule data type

Example

```
IC_Sched* sched;
sched = IC_Compute_schedule(myprog, prog, desc, region_set, 1);
```

3.3.4 IC_Send_TYPE

This function is used for sending a set of array regions to a remote application. There is a send function for each supported TYPE (char, short, int, float, double).

Synopsis

Fortran 90 IC_Send(to, sched, data, tag, status)
Note that the Fortran 90 IC_Send function is polymorphic, i.e., it does not require calling a particular version depending on the type of the **data** being received as do the C and Fortran 77 counterparts.

C int IC_Send_char(IC_Program* to, IC_Sched* sched, char* data, int tag)

Fortran IC_Send_char(to, sched, data, tag, status)
integer to, sched, tag, status
integer*1 data(*)

C int IC_Send_short(IC_Program* to, IC_Sched* sched, short* data, int tag)

Fortran IC_Send_short(to, sched, data, tag, status)
integer to, sched, tag, status
integer*2 data(*)

C int IC_Send_int(IC_Program* to, IC_Sched* sched, int* data, int tag)

Fortran IC_Send_int(to, sched, data, tag, status)
integer to, sched, tag, status
integer data(*)

C int IC_Send_float(IC_Program* to, IC_Sched* sched, float* data, int tag)

Fortran IC_Send_float(to, sched, data, tag, status)
integer to, sched, tag, status
real data(*)

C int IC_Send_double(IC_Program* to, IC_Sched* sched, double* data, int tag)

Fortran IC_Send_double(to, sched, data, tag, status)
integer to, sched, tag, status
real*8 data(*)

Parameters

to the InterComm application descriptor for the receiving program

sched the InterComm communication schedule

data the local array

tag a message tag for identification purposes

Return value

status, -1 indicates an error

Example

```
int sts, tag;
float* data = &A[0][0][0];
sts = IC_Send_float(prog, sched, data, tag);
```

3.3.5 IC_Recv_TYPE

This function is used for receiving a set of regions from a sending process. There is a receive function for each supported TYPE.

Synopsis

Fortran 90 IC_Recv(from, sched, data, tag, status)
Note that the Fortran 90 IC_Recv function is polymorphic, i.e., it does not require calling a particular version depending on the type of the **data** being received as do the C and Fortran 77 counterparts.

C int IC_Recv_char(IC_Program* from, IC_Sched* sched, char* data, int tag)

Fortran IC_Recv_char(from, sched, data, tag, status)
integer from, sched, tag, status
integer*1 data(*)

C int IC_Recv_short(IC_Program* from, IC_Sched* sched, short* data, int tag)

Fortran IC_Recv_short(from, sched, data, tag, status)
integer from, sched, tag, status
integer*2 data(*)

C int IC_Recv_int(IC_Program* from, IC_Sched* sched, int* data, int tag)

Fortran IC_Recv_int(from, sched, data, tag, status)
integer from, sched, tag, status
integer data(*)

C int IC_Recv_float(IC_Program* from, IC_Sched* sched, float* data, int tag)

Fortran IC_Recv_float(from, sched, data, tag, status)
integer from, sched, tag, status
real data(*)

C int IC_Recv_double(IC_Program* from, IC_Sched* sched, double* data, int tag)

Fortran IC_Recv_double(from, sched, data, tag, status)
integer from, sched, tag, status
real*8 data(*)

Parameters

from the sending program

sched the communication schedule

data the local array

tag a message tag

Return value

status, -1 indicates error

Example

```
int sts, tag;
float data[500][500][500];
tag = 99;
sts = IC_Recv_float(prog, sched, data, tag);
```

3.4 Releasing Library Resources

3.4.1 IC_Free_program

This function releases memory used for holding an InterComm application descriptor and disconnects it from the underlying communication infrastructure.

Synopsis

C void IC_Free_program(IC_Program* prog)

Fortran IC_Free_program(prog)
integer prog

Parameters

prog an InterComm data type representing a remote program

Return value

none

Example

```
IC_Free_program(prog);
```

3.4.2 IC_Free_desc

This function releases memory user for holding an InterComm array descriptor.

Synopsis

C void IC_Free_desc(IC_desc* desc)

Fortran IC_Free_desc(desc)
integer desc

Parameters

desc an InterComm distributed array descriptor

Return value

none

Example

```
IC_Free_desc(desc);
```

3.4.3 IC_Free_region

This function is used to release memory for holding an InterComm region descriptor.

Synopsis

C void IC_Free_region(IC_region* region)

Fortran IC_Free_region(region)
integer region

Parameters

region an InterComm region descriptor representing an array block

Return value

none

Example

```
IC_Free_region(region);
```

3.4.4 IC_Free_sched

This function releases memory for holding an InterComm communication schedule.

Synopsis

C void IC_Free_sched(IC_Sched* sched)

Fortran IC_Free_sched(sched)
integer sched

Parameters

sched an InterComm communication schedule

Return value

none

Example

```
IC_Free_sched(sched);
```

3.4.5 IC_Quit

Shuts down the communication subsystem and frees the local program data structure.

Synopsis

C int IC_Quit(IC_Program* myprog)

Fortran IC_Quit(myprog, status)
integer myprog, status

Parameters

myprog the local program

Return value

status, -1 indicates an error

Example

```
int sts;
sts = IC_Quit(myprog);
```

4 High-Level Programming Tasks

In some cases, the utilization of InterComm can be much simplified by relying on a few higher-level function calls. The functions described in Section **??** provide a generic way for initializing and finalizing InterComm, for defining array decompositions, for establishing communication between a pair of programs, among other tasks. On the other hand, many applications can make use of a simplified interface that encapsulates most of InterComm complexities. However, this high-level API, albeit being simpler, does not provide as much flexibility as the low-level interface. For example, in order to use the high-level interface array distributions must comply to a few *canonical* distributions as we will describe later in this document. The high-level API is available for C++ programs relying on the P++ library and also for *sequential* Fortran 90 programs.

4.1 Creating Endpoints

The high-level API relies on the concept of a *communication endpoint*. The endpoint is an abstraction that corresponds to a communication channel between a pair of programs that need to exchange arrays or array sections. The establishment of a communication endpoint is the first step to ensure that data can flow between a pair of programs.

4.1.1 EndPoint(Constructor)

Synopsis

C++ IC_EndPoint::IC_Endpoint(const char* endpointName, const unsigned mynproc,
const unsigned onproc, const unsigned myao, const unsigned oao, int& status)

Fortran 90 ic_endpointconstructor(icce, endpointName, mynproc, onproc, myao, oao, status)
ic_obj icce
character, (len=*) endpointName
integer mynproc, onproc, myao, oao, status

Parameters

icce *returns* a handle to the InterComm communication endpoint descriptor

endpointName the name for the endpoint
This must be in the format *first program* : *second program* (e.g., `simulation1:simulation2`). The Fortran 90 version requires explicitly appending a CHAR(0) to the end of the string.

mynproc the number of processors used by the local program

onproc the number of processors used by the remote (the other) program

myao the array *ordering* used by the local program
The valid inputs for this parameter are `IC_ROW_MAJOR` and `IC_COLUMN_MAJOR`.

oao the array *ordering* used by the remote (the other) program
The valid inputs for this parameter are `IC_ROW_MAJOR` and `IC_COLUMN_MAJOR`.

status *returns* the result of the endpoint creation operation

This result should always be checked to ensure that the endpoint is in sane state. In case of success, status is set to `IC_OK`.

Return value

The C++ call is a C++ *constructor call.* The Fortran version returns a *reference* to the endpoint in its *icce* parameter. Both calls return the status of the operation in the *status* parameter.

Example

C++:

```
IC_EndPoint right_left_ep("right:left",1,1,
    IC_EndPoint::IC_COLUMN_MAJOR,IC_EndPoint::IC_COLUMN_MAJOR,ic_err);
```

Fortran 90:

```
type(ic_obj) :: icce
call ic_endpointconstructor(icce,'left:right'//CHAR(0),1,1,
    IC_COLUMN_MAJOR,IC_COLUMN_MAJOR,ic_err)
```

4.2 Communicating Array Sections

Once the communication endpoint is created, the two applications can start exchanging data, by exporting and importing arrays or arrays subsections.

4.2.1 ExportArray

Synopsis

C++ IC_EndPoint::exportArray(const intArray& array, int& status)

Fortran 90 ic_exportarray(icce, array, status)
ic_obj icce
integer, dimension (:) array
integer status

C++ IC_EndPoint::exportArray(const floatArray& array, int& status)

Fortran 90 ic_exportarray(icce, array, status)
ic_obj icce
real, dimension (:) array
integer status

C++ IC_EndPoint::exportArray(const doubleArray& array, int& status)

Fortran 90 ic_exportarray(icce, array, status)
ic_obj icce
double precision, dimension (:) array
integer status

Parameters

icce the exporting InterComm communication endpoint handle

array the array or array section to be transfered

The C++ method and the Fortran 90 function call are both *polymorphic*, which means that the operation is correctly performed regardless of the type of the array elements.

status *returns* the result of the export operation

This result should always be checked to ensure that the operation was correctly performed. In case of success, status is set to `IC_OK`.

Return value

The C++ call is C++ *void* method invokation. The Fortran version also does not return a value. Both calls return the status of the operation by setting the *status* parameter.

Example

C++:

```
doubleArray DOUBLES(10,10);
Index I(3,6), J(1,4);
right_left_ep.exportArray(DOUBLES(J,I),ic_err);
```

Fortran 90:

```
double precision, dimension (10,10) :: DOUBLES
call ic_exportarray(icce,DOUBLES(2:5,4:9),ic_err);
```

4.2.2 ImportArray

Once an application starts exporting data, the other (remote) application is supposedly importing data.

Synopsis

C++ IC_EndPoint::importArray(const intArray& array, int& status)

Fortran 90 ic_importarray(icce, array, status)
ic_obj icce
integer, dimension (:) array
integer status

C++ IC_EndPoint::importArray(const floatArray& array, int& status)

Fortran 90 ic_importarray(icce, array, status)
ic_obj icce
real, dimension (:) array
integer status

C++ IC_EndPoint::importArray(const doubleArray& array, int& status)

Fortran 90 ic_importarray(icce, array, status)
ic_obj icce
double precision, dimension (:) array
integer status

Parameters

icce the importing InterComm communication endpoint handle

array the array or array section to be received

The C++ method and the Fortran 90 function call are both *polymorphic*, which means that the operation will be correctly performed regardless of the type of the array elements.

status *returns* the result of the import operation

This result should always be checked to ensure that the operation was correctly performed. In case of success, status is set to `IC_OK`.

Return value

The C++ call is C++ *void* method invokation. The Fortran version also does not return a value. Both calls return the status of the operation in the *status* parameter.

Example

C++:

```
floatArray FLOATS(10);
Index I(3,6);
right_left_ep.importArray(FLOATS(I),ic_err);
```

Fortran 90:

```
real, dimension (10) :: FLOATS
call ic_importarray(icce,FLOATS(2:5),ic_err);
```

4.3 Termination

When the pair of applications reach a point where data is no longer being exchanged, both applications are expected to destroy their end of the communication endpoint to ensure a clean shutdown of InterComm and also of the underlying communication infrastructure.

4.3.1 EndPoint(Destructor)

Synopsis

C++ IC_EndPoint::˜ IC_Endpoint()

Fortran 90 ic_endpointdestructor(icce)
ic_obj icce

Parameters

icce the InterComm endpoint to be shutdown

Return value

Note that the C++ call is an object destructor. It need not be called explicitly. In reality, the destructor is automatically called for deallocating communication endpoint objects statically created. The application writer is supposed to invoke the *delete* C++ operator to ensure that the destructor properly finalizes InterComm.

Example

C++:

```
IC_EndPoint* right_left_ep = new IC_EndPoint("right:left",1,1,
    IC_EndPoint::IC_COLUMN_MAJOR,IC_EndPoint::IC_COLUMN_MAJOR,ic_err);
delete right_left_ep; // indirectly invoking the object destructor
```

Fortran 90:

```
type(ic_obj) :: icce
call ic_endpointdestructor(icce)
```

4.4 Error Codes

All the high-level InterComm calls with the exception of the destructor call returns the status of the operation. For successful operations `IC_OK` is returned. For unsuccessful operations, a variety of error codes can be returned. A list of all possible return values is seen in Table 1.

InterComm provides an auxiliary function to help application developers handle erroneous operations. The following function call/method invokation can be employed to print out a message stating the nature of the error condition:

4.4.1 PrintErrorMessage

Synopsis

C++ IC_EndPoint::printErrorMessage(const char* msg, int& status)

Fortran 90 ic_printerrormessage(msg, status)
character (len=*) msg
integer status

Parameters

msg an error message
This string will precede the actual text corresponding to the error code. For the Fortran 90 call the string must have a CHAR(0) as the last character.

status the error code for which the error message will be printed out

Return value

The C++ call is C++ *void* method invokation. The Fortran version also does not return a value. The function prints out a message warning the user that the error code is invalid, if *status* does not hold any of the values displayed in Table 1.

Example

C++:

```
right_left_ep.printErrorMessage("API call failed",ic_err);
```

Fortran 90:

```
call printErrorMessage('API call failed'//CHAR(0),ic_err);
```

5 Compilation and Program Startup

A Example Code

A.1 Low-Level C

The following C program acts as a data sender application and uses a translation table for describing an array distribution.

```
#include <intercomm.h>
#include <pvm3.h>

IC_Desc* create_ttable(int p, int r, float A[200]) {
  int globals[200], offsets[200], tasks[200];
```

Error Condition	Error Message
IC_OK	no error
IC_GENERIC_ERROR	generic error
IC_INVALID_NDIM	invalid number of array dimensions
IC_CANT_ALLOC_REGION	can't allocate InterComm array regions
IC_CANT_GET_DA_DESCRIPTOR	can't obtain distributed array descriptor
IC_CANT_COMPUTE_COMM_SCHEDULE	can't compute communication schedule
IC_COMM_FAILURE	communication (send/recv) failure
IC_INVALID_ENDPOINT_NAME	invalid endpoint name
IC_INITIALIZATION_FAILURE	local program initialization failed
IC_CANT_CONNECT_TO_REMOTE	can't connect to remote program
IC_PVM_ERROR	PVM error
IC_MPI_ERROR	MPI error
IC_POINTER_TABLE_ERROR	internal pointer translation table error – possibly too many InterComm descriptors (regions, distributions, etc) have been defefined

Table 1: InterComm high-level API errors

```
  IC_Desc* desc;
  int i;

  /* assign this processor the rth portion of global index space */
  for (i = 0; i < 200; ++i) {
    globals[i] = 200*r + i;
  }

  /* round-robin global index to task assignment */
  for (i = 0; i < 200; ++i) {
    offsets[i] = i/p + 50*r;
    tasks[i] = i%p;
  }

  /* create the ic descriptor */
  desc = IC_Create_ttable_desc(globals, offsets, tasks, 200);

  /* assign each local value its global index */
  for (i = 0; i < 200; ++i) {
    A[i] = globals[i];
  }

  return desc;
}
```

```
int main(int argc, char* argv[]) {
  char* localname = "cexample";
  char* othername = "fexample";
  int local_tasks = 4;
  int other_tasks = 8;
  int i, rank;

  char* groupcomm = "local_c";

  IC_Program* local;
  IC_Program* other;

  IC_Desc* desc;
  float A[200];

  IC_Region* region_set[2];
  int indices[8];
  IC_Sched* sched;
  int tag = 99;

  /* initialize pvm */
  printf("initialize pvm\n");
  rank = pvm_joingroup(groupcomm);
  pvm_barrier(groupcomm, local_tasks);

  /* initialize ic */
  printf("initialize ic\n");
  local = IC_Init(localname, local_tasks, rank);
  other = IC_Wait(othername, other_tasks);
  IC_Sync(local, other);

  desc = create_ttable(local_tasks, rank, A);

  /* define two 8 element regions for transfer */
  printf("define regions\n");
  for (i = 0; i < 8; ++i)
    indices[i] = i;
  region_set[0] = IC_Create_enum_region(indices, 8);
  for (i = 0; i < 8; ++i)
    indices[i] = i + 400;
  region_set[1] = IC_Create_enum_region(indices, 8);

  /* create a translation table type array descriptor and array */
  printf("create schedule\n");
  sched = IC_Compute_schedule(local, other, desc, region_set, 2);
  printf("send data\n");
```

```
  IC_Send_float(other, sched, &A[0], tag);
  IC_Sync(local, other);
  IC_Free_sched(sched);

  /* clean up */
  printf("clean up\n");
  IC_Free_region(region_set[1]);
  IC_Free_region(region_set[0]);

  IC_Free_desc(desc);

  IC_Free_program(other);
  IC_Quit(local);

  pvm_barrier(groupcomm, local_tasks);
  pvm_lvgroup(groupcomm);
  pvm_exit();

  return 0;
}
```

A.2 Low-Level Fortran

The following program is an application that acts as the receiver end of a data exchange operation and uses a regular block decomposition.

```
      program fexample
      implicit none
      include 'fpvm3.h'

      character*8 localname
      data localname / 'fexample' /
      character*8 othername
      data othername / 'cexample' /
      integer local_tasks
      data local_tasks / 8 /
      integer other_tasks
      data other_tasks / 4 /
      integer i, rank

      character*7 groupcomm
      data groupcomm / 'local_f' /

      integer local
```

```
      integer other

      integer desc
      real A(5,5,5)

      integer region_set(2)
      integer lower(3), upper(3), stride(3)
      integer sched
      integer tag
      data tag / 99 /

      integer sts

c     initialize pvm
      print *, 'initialize pvm'
      call pvmfjoingroup(groupcomm, rank)
      call pvmfbarrier(groupcomm, local_tasks, sts)

c     initialize ic
      print *, 'initialize ic'
      call IC_Init(localname, local_tasks, rank, local)
      call IC_Wait(othername, other_tasks, other)
      call IC_Sync(local, other, sts)

c     create a block decomposition type array descriptor and array
      call create_bdecomp(local_tasks, rank, A, desc)

c     define two 2x2x2 regions for transfer
      print *, 'define regions'
      do i = 1, 3
         lower(i) = 1
         upper(i) = 2
         stride(i) = 1
      end do
      call IC_Create_block_region(3, lower, upper, stride,
     $     region_set(1))
      do i = 1, 3
         lower(i) = 6
         upper(i) = 7
         stride(i) = 1
      end do
      call IC_Create_block_region(3, lower, upper, stride,
     $     region_set(2))

      print *, 'create schedule'
      call IC_Compute_schedule(local, other, desc, region_set,
```

```
     $     2, sched)
      print *, 'receive data'
      call IC_Recv_float(other, sched, A, tag, sts)
      call IC_Sync(local, other, sts)
      call IC_Free_sched(sched)

c     cleanup
      print *, 'clean up'
      call IC_Free_region(region_set(2))
      call IC_Free_region(region_set(1))

      call IC_Free_desc(desc)

      call IC_Free_program(other)
      call IC_Quit(local, sts)

      call pvmfbarrier(groupcomm, local_tasks, sts)
      call pvmflvgroup(groupcomm, sts)
      call pvmfexit(sts)

      end


      subroutine create_bdecomp(p, r, A, desc)
      implicit none
      integer p, r
      real A(5,5,5)
      integer desc

      integer blocks(8,2,3)
      integer tasks(8)
      integer i, j, k

c     bisect the global array in each dimension
      do i = 0, 1
         do j = 0, 1
            do k = 0, 1
               blocks(4*i+2*j+k+1,1,1) = i*5 + 1
               blocks(4*i+2*j+k+1,2,1) = i*5 + 5
               blocks(4*i+2*j+k+1,1,2) = j*5 + 1
               blocks(4*i+2*j+k+1,2,2) = j*5 + 5
               blocks(4*i+2*j+k+1,1,3) = k*5 + 1
               blocks(4*i+2*j+k+1,2,3) = k*5 + 5
            end do
         end do
      end do
```

```
c     assign the blocks to the tasks
      do i = 1, 8
         tasks(i) = i - 1
      end do

c     create the ic descriptor
      call IC_Create_bdecomp_desc(3, blocks, tasks, 8, desc)

c     assign each local element a number based on its index
      do i = 1, 5
         do j = 1, 5
            do k = 1, 5
               A(i,j,k) = 1000*r + 100*i + 10*j + k
            end do
         end do
      end do

      end
```

A.3 High-Level C++/P++

In this section, we show the source code for two programs that exchange arrays and array sections bidirectionally. The programs are called "left" – a C++/P++ program shown in Section A.3 – and "right" – a sequential Fortran 90 program shown in Section A.4.

```
// This code shows how data can be exchanged (imported/exported) between a
// Fortran 90 program and a C++ program using P++

// Note that P++ uses column-major for representing the data, but uses the
// C-style indexing schema for the array elements. That is a(i,j) in Fortran
// is actually a(j,i) in C++/P++
#include <iostream>
#include <iomanip>
#include <stdlib.h>
// using P++
#include "A++.h"
// using InterComm high-level library
#include "IC_EndPoint.h"

int main(int argc, char **argv) {
  int iNumProcs=1;
  // initializing P++ virtual machine
  Optimization_Manager::Initialize_Virtual_Machine("",iNumProcs,argc,argv);
```

```
int ic_err;
// creating communication endpoint
IC_EndPoint right_left_ep("right:left",iNumProcs,1,
                          IC_EndPoint::IC_COLUMN_MAJOR,
                          IC_EndPoint::IC_COLUMN_MAJOR,
                          ic_err);
// testing if the endpoint has been properly created
if (ic_err!=IC_EndPoint::IC_OK) {
  IC_EndPoint::printErrorMessage("ic_endpointconstructor",ic_err);
  return ic_err;
}

// array declarations
intArray INTEGERS(10);
doubleArray DOUBLES(10,9);
floatArray FLOATS(3,11,10);

// array range declarations
Index I(3,6), J(1,4), K(1,2);

// array initialization operations
INTEGERS=33;
FLOATS=44;

unsigned i, j;
for(i=0;i<10;i++) {
  for(j=0;j<9;j++) {
    DOUBLES(i,j)=100+(10*i)+j;
  }
}

// playing with 1D array of integers
cout << "before receiving data:" << endl;
PRINT_1D_ARRAY(INTEGERS);
// importing array section from the "left" side program
right_left_ep.importArray(INTEGERS(I),ic_err);
assert(ic_err==IC_EndPoint::IC_OK);
cout << "after receiving data:" << endl;
PRINT_1D_ARRAY(INTEGERS);

// playing with 2D array of doubles
cout << "before receiving data:" << endl;
PRINT_2D_ARRAY(DOUBLES);

// importing array section from the "left" side program
```

```
  right_left_ep.importArray(DOUBLES(J,I),ic_err);
  assert(ic_err==IC_EndPoint::IC_OK);

  cout << "after receiving data:" << endl;
  PRINT_2D_ARRAY(DOUBLES);

  DOUBLES(J,I)=10;
  cout << "before exporting data:" << endl;
  PRINT_2D_ARRAY(DOUBLES);
  // exporting array section to the "left" side program
  right_left_ep.exportArray(DOUBLES(J,I),ic_err);
  assert(ic_err==IC_EndPoint::IC_OK);

  // playing with 3D array of floats
  cout << "before receiving data:" << endl;
  PRINT_3D_ARRAY(FLOATS);

  // importing array section from the "left" side program
  right_left_ep.importArray(FLOATS(K,J,I),ic_err);
  assert(ic_err==IC_EndPoint::IC_OK);

  cout << "after receiving data:" << endl;
  PRINT_3D_ARRAY(FLOATS);

  FLOATS=55;
  cout << "before exporting data:" << endl;
  PRINT_3D_ARRAY(FLOATS(K,J,I));
  // exporting array section to the "left" side program
  right_left_ep.exportArray(FLOATS(K,J,I),ic_err);
  assert(ic_err==IC_EndPoint::IC_OK);

  Optimization_Manager::Exit_Virtual_Machine();
  return 0;
}
```

A.4 High-Level Fortran 90

```
! This code shows how data can be exchanged between a Fortran 90 program and a
! C++ program using P++
program intercomm_test

! using InterComm high-level API
  use intercomm_interface
  use array_printing
```

```
  implicit none

! array and variable declarations
  integer, dimension (10) :: INTEGERS
  ! 10 rows and 9 columns
  double precision, dimension (10,9) :: DOUBLES
  ! 3 2-D matrices each with 11 rows and 10 columns
  real, dimension (3,11,10) :: FLOATS
  type(arraydesc) :: FLOATSDesc
  integer :: i, j, ic_err

  type(ic_obj) :: icce

  ! initialize small array
  do i = 1, 10
    INTEGERS(i) = 10*(i-1)
    do j = 1, 9
      DOUBLES(i,j) = 10*(i-1)+(j-1)
    end do
  end do

! creating InterComm communication endpoint
  call ic_endpointconstructor(icce,'left:right'//CHAR(0),1,1,
                             IC_COLUMN_MAJOR,IC_COLUMN_MAJOR,ic_err);
! testing if communication endpoint is in sane state
  if (ic_err <> IC_OK) then
    call ic_printerrormessage('ic_endpointconstructor'//CHAR(0),ic_err)
    stop
  end if

  print *,">>>> dealing with a 1D array of integers"
!!!!!!!!!!!!!!!!!!!!!! dealing with a 1D array of integers
  call printArray(INTEGERS)
! exporting array to "right" side application
  call ic_exportarray(icce,INTEGERS(4:9),ic_err);
  if (ic_err <> IC_OK) then
    call ic_printerrormessage('ic_endpointdestructor'//CHAR(0),ic_err)
  end if

  print *,">>>> dealing with a 2D double precision array"
!!!!!!!!!!!!!!!!!!!!!! dealing with a 2D double precision array
  call printArray(DOUBLES)
  print *,"subarray"
! communicating parts of the DOUBLES array
  call ic_dumparrayinfo(DOUBLES(2:5,4:9));
```

```
  call printArray(DOUBLES(2:5,4:9));
! exporting array to "right" side application
  call ic_exportarray(icce,DOUBLES(2:5,4:9),ic_err);
  if (ic_err <> IC_OK) then
    call ic_printerrormessage('ic_endpointdestructor'//CHAR(0),ic_err)
  end if
  call ic_importarray(icce,DOUBLES(2:5,4:9),ic_err);
  if (ic_err <> IC_OK) then
    call ic_printerrormessage('ic_endpointdestructor'//CHAR(0),ic_err)
  end if
  print *,"after import"
  call printArray(DOUBLES)

!!!!!!!!!!!!!!!!!!!!!! dealing with a 3D array of floats
  print *,"before receiving data:"
!  call ic_dumparrayinfo(FLOATS(2:3,1:4,3:8));
  call ic_dumparrayinfo(FLOATS);
  FLOATSdesc = createArrayDesc(FLOATS)
  call printRealArray(FLOATSdesc)
! exporting array to "right" side application
  call ic_exportarray(icce,FLOATS(2:3,1:4,3:8),ic_err);
  if (ic_err <> IC_OK) then
    call ic_printerrormessage('ic_endpointdestructor'//CHAR(0),ic_err)
  end if
! importing array from "right" side application
  call ic_importarray(icce,FLOATS(2:3,1:4,3:8),ic_err);
  if (ic_err <> IC_OK) then
    call ic_printerrormessage('ic_endpointdestructor'//CHAR(0),ic_err)
  end if
  print *,"after receiving data:"
  call printRealArray(FLOATSdesc)

! destroying communication endpoint
  call ic_endpointdestructor(icce);

  print *,"done"

end program intercomm_test
```

References

[1] Object-Oriented Tools for Solving PDEs in Complex Geometries, http://www.llnl.gov/CASC/Overture/

[2] PVM: Parallel Virtual Machine, http://www.csm.ornl.gov/pvm/pvm_home.html

[3] GNU Autoconf, Automake, and Libtool, http://www.gnu.org/directory/

[4] Chaos Tools, http://www.cs.umd.edu/projects/hpsl/chaos/ResearchAreas/tools.html

[5] MultiBlockParti, http://www.cs.umd.edu/projects/hpsl/chaos/ResearchAreas/tools.html

[6] The Message Passing Interface (MPI) Standard, http://www-unix.mcs.anl.gov/mpi/